Memory Characterization of Workloads Using Instrumentation-Driven Simulation

A Pin-based Memory Characterization of the SPEC CPU2000 and SPEC CPU2006 Benchmark Suites

Aamer Jaleel

Intel Corporation, VSSAD
{aamer.jaleel}@intel.com

Abstract

There is a growing need for simulation methodologies to understand the memory system requirements of emerging workloads in a reasonable amount of time. This paper presents binary instrumentation-driven simulation as an alternative to conventional execution-driven and trace-driven simulation methodologies. We illustrate the use of instrumentation-driven simulation (IDS) using Pin to determine the memory system requirements of workloads from the SPEC CPU2000 and SPEC CPU2006 benchmark suites. In comparison to SPEC CPU2000, SPEC CPU2006 workloads are an order of magnitude longer (in terms of instruction count). Additionally, SPEC CPU2006 comprises of many more memory intensive workloads that require more than 4MB of cache size for better cache performance.

1. Introduction

Processor architects continue to face key design decisions in designing the memory hierarchy. With several emerging application domains, understanding the memory system requirements of new applications is essential in designing an efficient memory hierarchy. Such characterization and exploratory studies require fast simulation techniques that can compare and contrast the performance of alternative design policies. For memory system studies, this paper proposes *instrumentation driven simulation* as an alternative to existing execution-driven and trace-driven methodologies.

Simulation is a common methodology that is used to identify performance bottlenecks in existing systems as well as design space exploration. There exist many simulators and software tools to investigate the memory system performance of applications. In general, memory system simulators fall into two main categories: trace-driven or execution-driven. With trace-driven memory system simulation, address traces are fed off-line to a memory system simulator (e.g. Dinero IV [3]). Such simulators rely on existing tools to collect memory address traces and log them to file for later use. Execution-driven cache simulators on the other hand rely on functional/performance models to execute application binaries. The memory addresses generated by the functional/performance model are fed in real time to a memory system simulator modeled within the functional/performance model. In general, functional models of modern ISAs are slow and can be complex to build. As a result, trace-driven simulation has become popular for conducting memory system studies [15].

The usefulness of trace-driven simulation, however, lies in the continued availability of address traces to study the memory system requirements of different workloads. With several emerging application domains, understanding the memory behavior and working set requirements of new applications requires the ability to generate address traces. Address trace generation for a target ISA can require sophisticated hardware tools (e.g. bus tracer) or a functional model that supports the target ISA and the requirements of the workload (e.g. the functional model must provide support for multiple contexts if executing a multi-threaded workload). However, there are drawbacks with these mechanisms.

Hardware tools such as bus tracers only capture address traces that reach the system bus. Consequently, the address trace collected is incomplete since address requests that hit in the processor caches are filtered out. On the other hand, address trace generation using function models represent only a small region of the application. Additionally, a practical problem with collecting memory address traces with either mechanism is that they can become very large, potentially occupying several gigabytes of disk space even in their compressed formats.

To address the drawbacks of current simulation techniques, this paper proposes the use of *instrumentation driven simulation (IDS)* to conduct memory system studies. Since instrumentation-driven memory system simulation is *fast*, *robust*, and *simple*, users can write simple tools to characterize the memory system requirements of almost *any* application at MIPS (as opposed to KIPS) speed. In doing so, IDS can support *full run* application studies.

This paper presents the use of Pin [7] based IDS to conduct full-run memory system studies of workloads from the SPEC CPU2000 and SPEC CPU2006 [4] benchmark suites. Our studies reveal that on average, workloads in SPEC CPU2006 have an instruction count that is an order of magnitude larger than the SPEC CPU2000 workloads. Additionally, SPEC CPU2006 workloads have larger memory working-set sizes

with most memory-intensive workloads requiring more than 4MB of cache size.

The rest of this paper is organized as follows. Section 2 provides the benefits and caveats of using IDS. Section 3 describes the simulation methodology and then presents a memory performance comparison of the SPEC CPU2000 and SPEC CPU2006 workloads. Finally Section 4 provides a summary of our work.

2. Instrumentation-Driven Simulation (IDS)

Binary instrumentation is a technique for inserting extra code into an existing application to collect run time information from an application. The binary instrumentation tool determines the type of run time information to be collected. Typically, binary instrumentation tools have commonly been used for the performance analysis of applications. However, binary instrumentation systems can also serve as an attractive tool for conducting computer architecture research studies. For example, binary instrumentation tools can model simple simulators that are *instrumentation-driven*.

In addition to being simple, binary instrumentation systems are *fast* and *robust*. Since binary instrumentation normally occurs at native execution speeds, IDS can also occur at MIPS speed. Unlike existing execution and trace-driven simulators, IDS supports quick exploratory studies by simulating applications run to completion. Additionally, since binary instrumentation systems are robust for all kinds of applications, users can conduct instrumentation-driven simulation with *any* kind of application, no matter its complexity. For example, users can study complex applications such as Oracle or Java.

2.1. Caveats with Instrumentation-Driven Simulation

As with any simulation methodology there are some tradeoffs with the instrumentation-driven approach.

- ***Instrumentation Overhead:*** Instrumentation involves injecting extra code dynamically or statically into the target application. The additional code causes an application to spend extra time in executing the original application. If the application has the property that it changes its behavior based on the amount of time spent in execution, the native execution and instrumented execution of the application can be different. If the differences are significant, IDS is unsuitable for such class of applications. Additionally, for multi-threaded applications, instrumentation can modify the ordering of instructions executed between different threads of the application. As a result, IDS with multi-threaded applications comes at the lack of some fidelity.
- ***Lack of Speculation*:** Instrumentation only observes instructions executed on the correct path of execution. As a result, IDS may not be able to support wrong-path execution. If speculation is desired, special purpose tools to emulate a branch predictor can be implemented [11].
- ***Repeatability:*** Since instrumentation occurs on real operating systems, multiple runs of the same workload on the same system are not identical. If repeatability is desired, special purpose tools can be implemented [10].
- ***User-level Traffic Only:*** Current binary instrumentation tools only support user-level instrumentation. Thus, applications that are kernel intensive are unsuitable for user-level IDS. For such applications system level simulators [6, 8] or instrumentation systems [2] are ideal.
- ***Native System Limited:*** Since IDS is conducted on a native system, the fidelity of the simulation model can depend on the characteristics of the native system. For example, simulating a system that has four active hardware threads on a system that only supports two active hardware threads may incorrectly interleave the order in which instructions are executed between different hardware threads. This is because the native system time slices the simulated threads while a real simulated system executes the threads simultaneously. To ensure correct timing, this issue can be resolved by modeling a thread scheduler [11].

Despite the above mentioned caveats, IDS is an excellent methodology for initial exploratory studies as it is *fast, robust,* and *simple*.

3. Memory Behavior of SPEC CPU2000 and SPEC CPU2006

For purposes of this study, we use IDS for characterizing the memory system behavior of both the SPEC CPU2000 and SPEC CPU2006 benchmark suites. The workloads in the SPEC suite are attractive for IDS as they are unaffected by the caveats mentioned in the previous section. Existing work has studied the memory behavior of SPEC CPU2000[13]. For the purpose of uniformity in ISA, compilers and simulation methodology, we present a comprehensive cache performance study of all workloads from both SPEC CPU2000 and SPEC CPU2006 *run to completion*. We first describe the experimental methodology before presenting our results.

3.1. Experimental Methodology

We use Pin, a dynamic binary instrumentation system for Linux, MacOS, and Windows binary executables for Intel® IA-32 (x86 32-bit), IA-32E (x86 64-bit), and Itanium® processors. Pin is similar to the ATOM[14] toolkit for Alpha processors. Like ATOM, Pin provides an infrastructure for writing program analysis tools called *pin tools*.

For our workload characterization studies we use CMP$im [5], an instrumentation driven system simulator for CMPs. CMP$im can characterize application instruction profiles and conduct full run cache performance studies of applications at

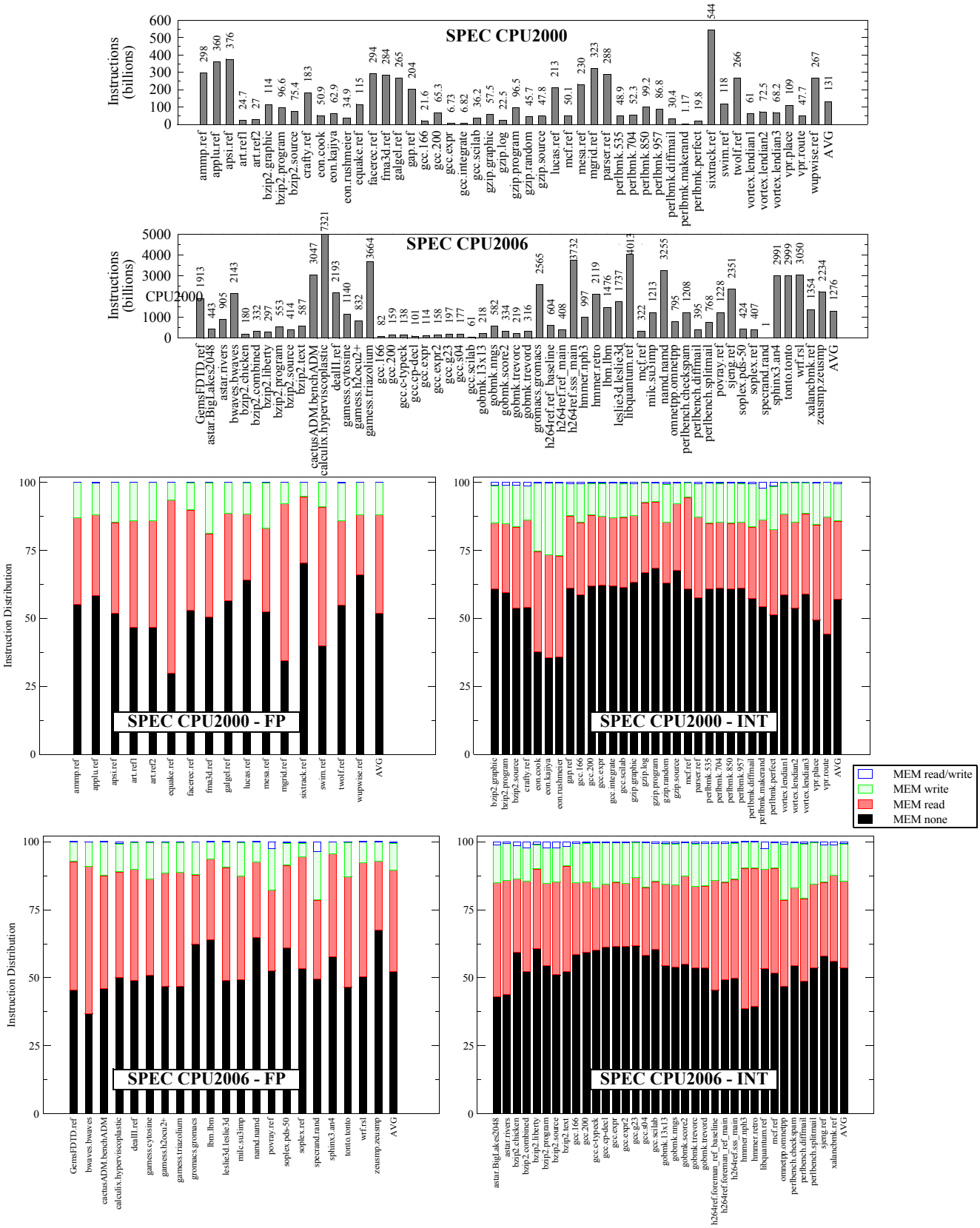


Figure 1: Dynamic Instruction Count and Instruction Profile of SPEC CPU2000 and SPEC CPU2006 workloads. The figure shows the instruction count and instruction mix of the applications when run to completing using the reference input sets.

the speeds of 4-10 MIPS. We use CMP$im in single-core mode to conduct detailed characterization studies of the SPEC workloads. All workloads from the SPEC CPU2000 and SPEC CPU2006 benchmark suites are run to completion using their reference input sets. We also collect IPC numbers for the two suites using performance counters on an Intel® Xeon® 2.8 GHz system with a two-level cache hierarchy: 32KB L1 cache and 512KB L2 cache. All workloads are compiled using optimization flags -O3 on a Red Hat 32-bit Linux system using Intel's C/C++ and Fortran compilers.

For each workload, we conduct a cache sensitivity analysis using the stack distance [9] approach to simulate multiple cache sizes in one run. We modeled a 256-way 8MB instruction cache and a 2048-way 128MB data cache. Both use 64B line size and model true LRU replacement policy. The simulated cache configurations support cache sizes that are direct mapped 32KB, 2-way 64KB, 4-way 128KB, and so on. After presenting the cache sensitivity study, we also present the cache performance of each workload for a three level cache hierarchy: 4-way 32KB separate L1 instruction and data caches, a unified 8-way 256KB L2 cache, and finally a unified 16-way 2MB L3 cache. All caches in the simulated hierarchy use 64B line size, are non-inclusive, allocate on writes, and use writeback and true LRU replacement policy.

3.2. Application Instruction Profile

Figure 1 shows the total instruction count for each workload in the two suites. The average instruction count for the SPEC CPU2000 benchmark suite is 131 billion while that of SPEC CPU2006 is 1276 billion—an order of magnitude higher. The increase in instruction count by SPEC has been in response to significant differences observed in run time as a result of small fluctuations in the system state or measurement conditions [1]. However, the large run lengths of SPEC CPU2006 now present interesting challenges in choosing representative regions of the workloads for conducting experiments using detailed performance simulators.

To understand the contribution of instructions that reference memory, Figure 1 presents the instruction profile for the floating-point and integer benchmarks distributed into four categories: instructions that do not reference memory (ALU operations only), instructions that have one or more source operands in memory (MEM read), instructions whose destination operand is in memory (MEM write), and instructions whose source and destination operands are in memory (MEM read and write). On average, roughly half of the instructions reference memory. Of the instructions that reference memory, very few (< 1%) both read from and write to memory while 20% of total memory instructions write to memory. This behavior is consistent for both SPEC CPU2000 and SPEC CPU2006. The large proportion of instructions that reference memory is indicative of register spills to the stack due to the limited registers available on 32-bit x86 systems.

3.3. Processor Performance

To identify the memory bound workloads, Figure 4 presents the CPI values of the two suites collected using performance counters on the 2.8 GHz Xeon system. The average CPI of the SPEC CPU2000 workloads is 1.97 while that of SPEC CPU2006 is 2.16. The higher CPI of SPEC CPU2006 is a result of larger problem set sizes [1]. Table 1 categorizes workloads based on their CPI values: CPI values greater than 8, CPI values greater than 4 but less than 8, and CPI values greater than 2 but less than 4. The table shows that SPEC CPU2006 represents more applications that are memory bound (higher CPI values) than SPEC CPU2000. The primary memory bound SPEC CPU2000 workloads (with CPIs greater than 4) were *art*, *equake*, *mcf*, *swim*, and *vpr* (route input). Comparatively, the memory bound SPEC CPU2006 workloads are: *GemsFDTD*, *astar* (BigLakes2048 input), *cactusADM*, *lbm*, *leslie3d*, *libquantum*, *mcf*, *milc*, *omnetpp*, and *soplex*. To understand the memory system requirements of these workloads, the next section presents workload sensitivity to different cache sizes.

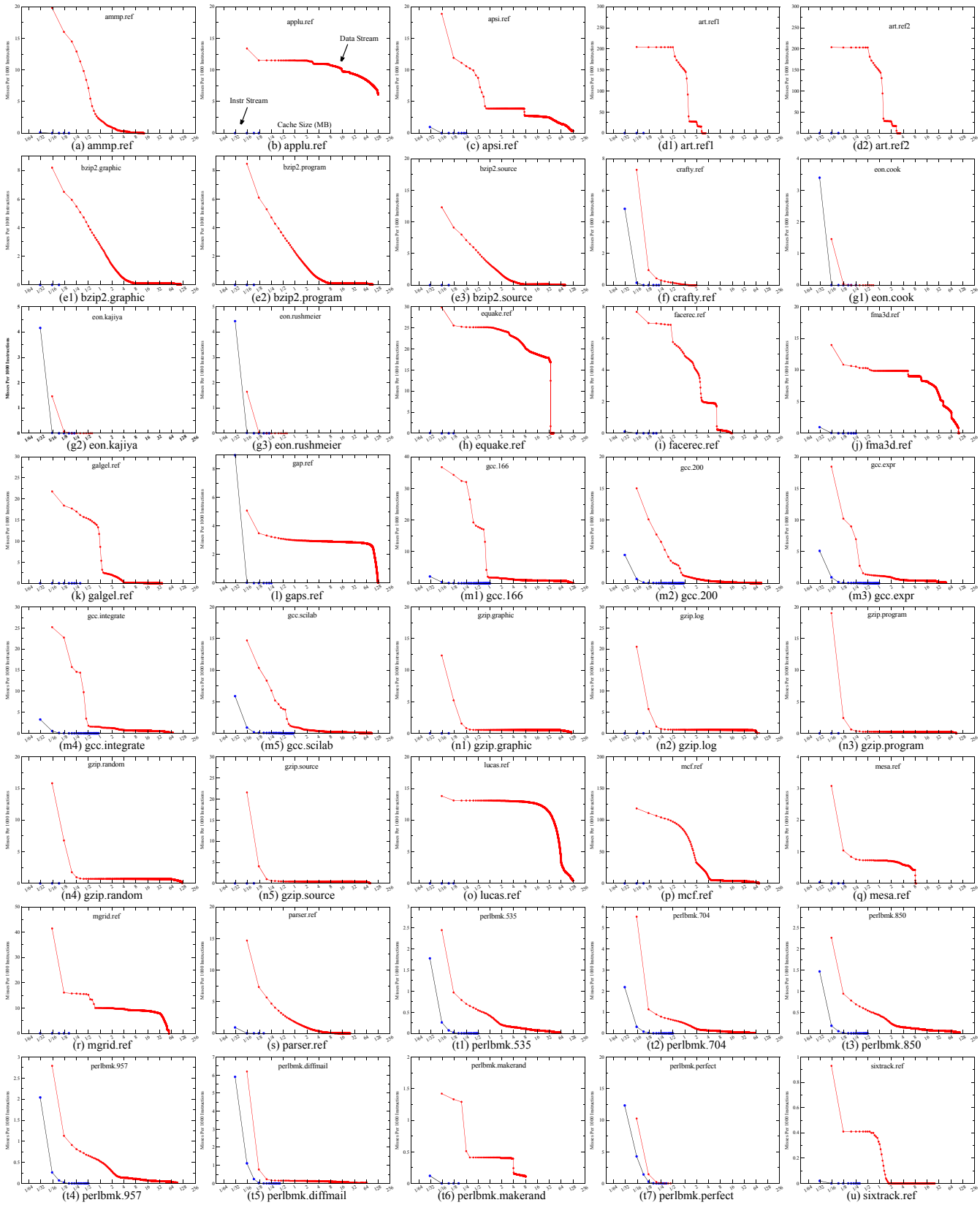
3.4. Cache Performance

We study working set analysis of the workloads by studying their cache performance with different cache sizes. We then present the cache performance of the SPEC workloads for a three-level cache hierarchy that is representative of modern high performance processors.

3.4.1.Working Set Size

Figures 2-3 and 5-6 present the instruction and data cache sensitivity studies for all workloads in the two suites. The figures present cache size in megabytes (MB) on a logarithmic x-axis and misses per one thousand (MPKI) instructions on the y-axis. Note that the y-axis varies significantly between different workloads.

We use a stack distance approach to simulate the performance of multiple cache sizes in one simulation run by using a single large cache. In doing so, the stack distance approach not only provides cache performance for multiple cache sizes but also provides an indication of the total memory footprint of each workload. For example, if the workload has a total instruction and data footprint that is smaller than the simulated large cache, the amount of valid data in the cache (assuming the cache is invalid at simulation start) represents the total memory footprint of the workload. This information is represented on the cache sensitivity graphs for workloads where the last x-axis co-ordinate ends before the simulated large cache size (8MB for instruction and 128MB for data). For example, the last x-coordinate data cache point for *art* (Figure 2d) occurs at 4MB, implying that it has a total data memory footprint size of approximately 4MB. Similarly, *art* has an instruction memory footprint size of approximately



(a) ammp.ref (b) applu.ref (c) apsi.ref (d1) art.ref1 (d2) art.ref2
(e1) bzip2.graphic (e2) bzip2.program (e3) bzip2.source (f) crafty.ref (g1) eon.cook
(g2) eon.kajiya (g3) eon.rushmeier (h) equake.ref (i) facerec.ref (j) fma3d.ref
(k) galgel.ref (l) gaps.ref (m1) gcc.166 (m2) gcc.200 (m3) gcc.expr
(m4) gcc.integrate (m5) gcc.scilab (n1) gzip.graphic (n2) gzip.log (n3) gzip.program
(n4) gzip.random (n5) gzip.source (o) lucas.ref (p) mcf.ref (q) mesa.ref
(r) mgrid.ref (s) parser.ref (t1) perlbmk.535 (t2) perlbmk.704 (t3) perlbmk.850
(t4) perlbmk.957 (t5) perlbmk.diffmail (t6) perlbmk.makerand (t7) perlbmk.perfect (u) sixtrack.ref

Figure 2: SPEC CPU2000 Cache Sensitivity: The figure shows the cache performance of each workload as a function of cache size. The y-axis presents workload MPKI and the x-axis presents the cache size in megabytes (MB). All workloads were run to completion using the reference input sets.

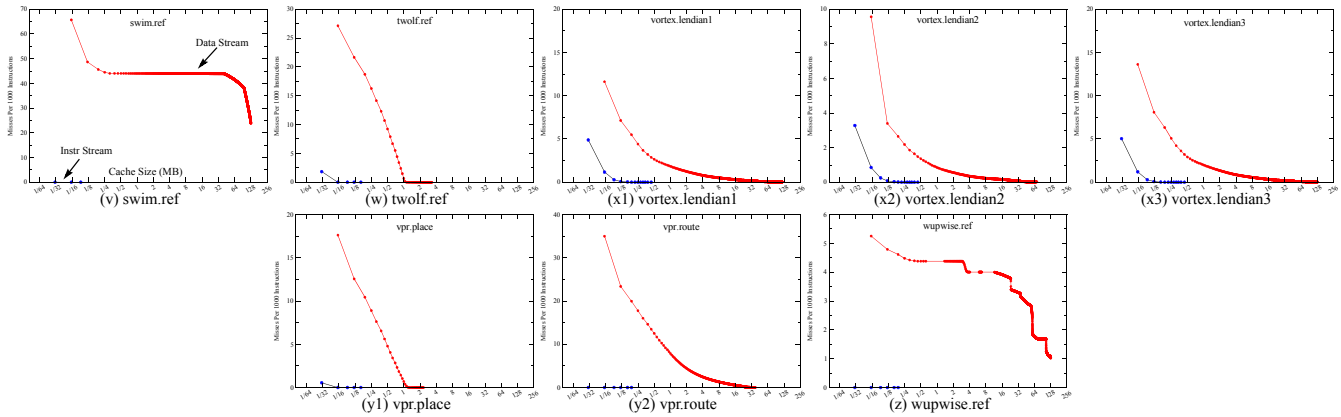


Figure 3: SPEC CPU2000 Cache Sensitivity: The figure shows the cache performance of each workload as a function of cache size. The y-axis presents workload MPKI and the x-axis presents the cache size in megabytes (MB). All workloads were run to completion using the reference input sets.

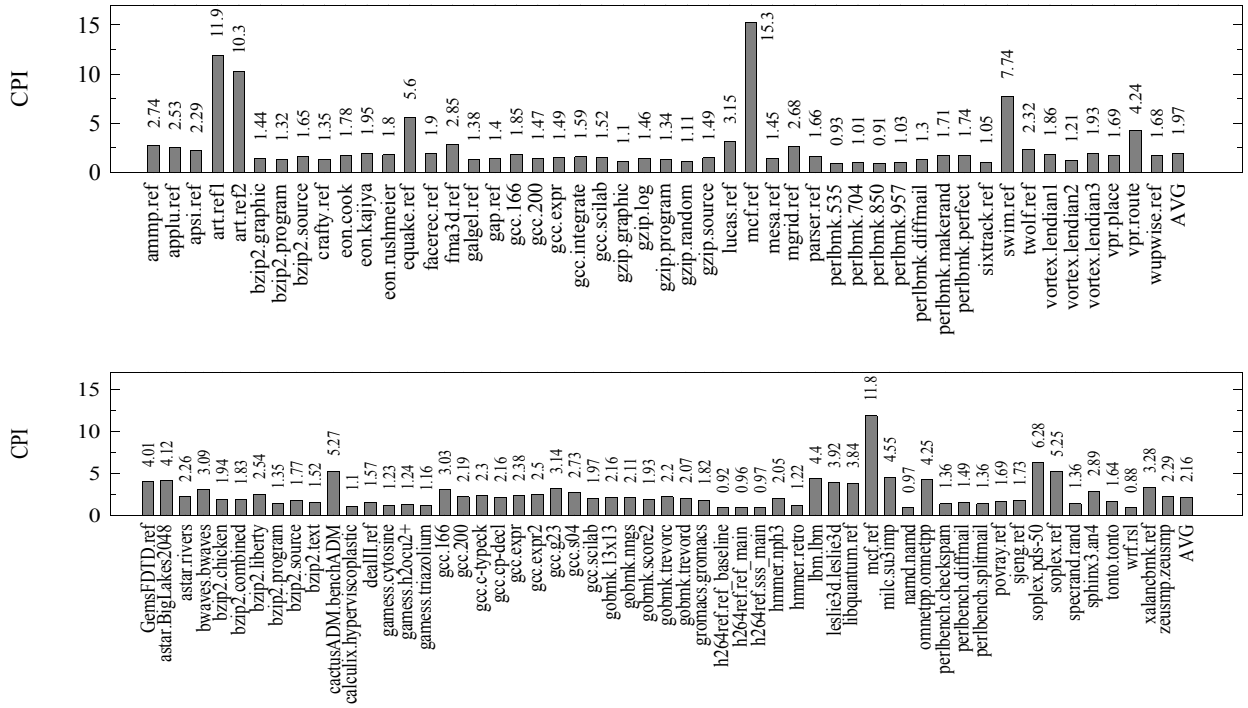


Figure 4: Performance of SPEC CPU2000 and SPEC CPU2006 Workloads: The figure shows the CPI values for the two suites gathered using performance counters on a 2.8GHz Xeon system. The table below presents the memory bounds workloads separated into three categories.

Table 1: Memory Bound Workloads

CPI	SPEC CPU2000	SPEC CPU2006
> 8	art, mcf	mcf
4 - 8	equake, swim, vpr.route	GemsFDTD, astar.BigLakes2048, cactusADM, lbm, milc, omnetpp, soplex
2 - 4	ammp, applu, apsi, fma3d, lucas, mgrid, twolf	astar.rivers, bwaves, bzip2.liberty, gcc, gobmk, hmmer.nph3, leslie3d, libquantum, sphinx3, xalancbmk, zeusmp

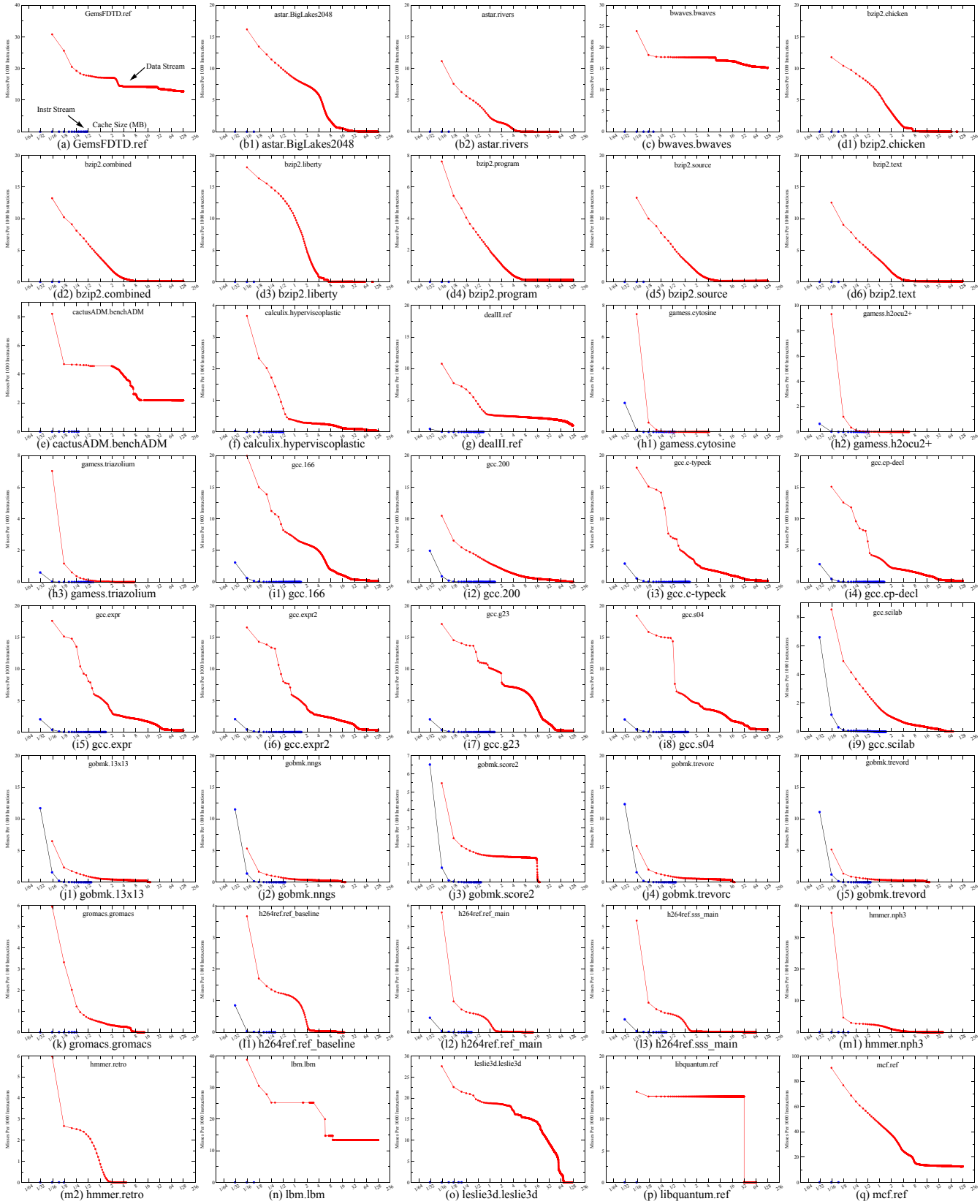


Figure 5: SPEC CPU2006 Cache Sensitivity: The figure shows the cache performance of each workload as a function of cache size. The y-axis presents workload MPKI and the x-axis presents the cache size in megabytes (MB). All workloads were run to completion using the reference input sets.

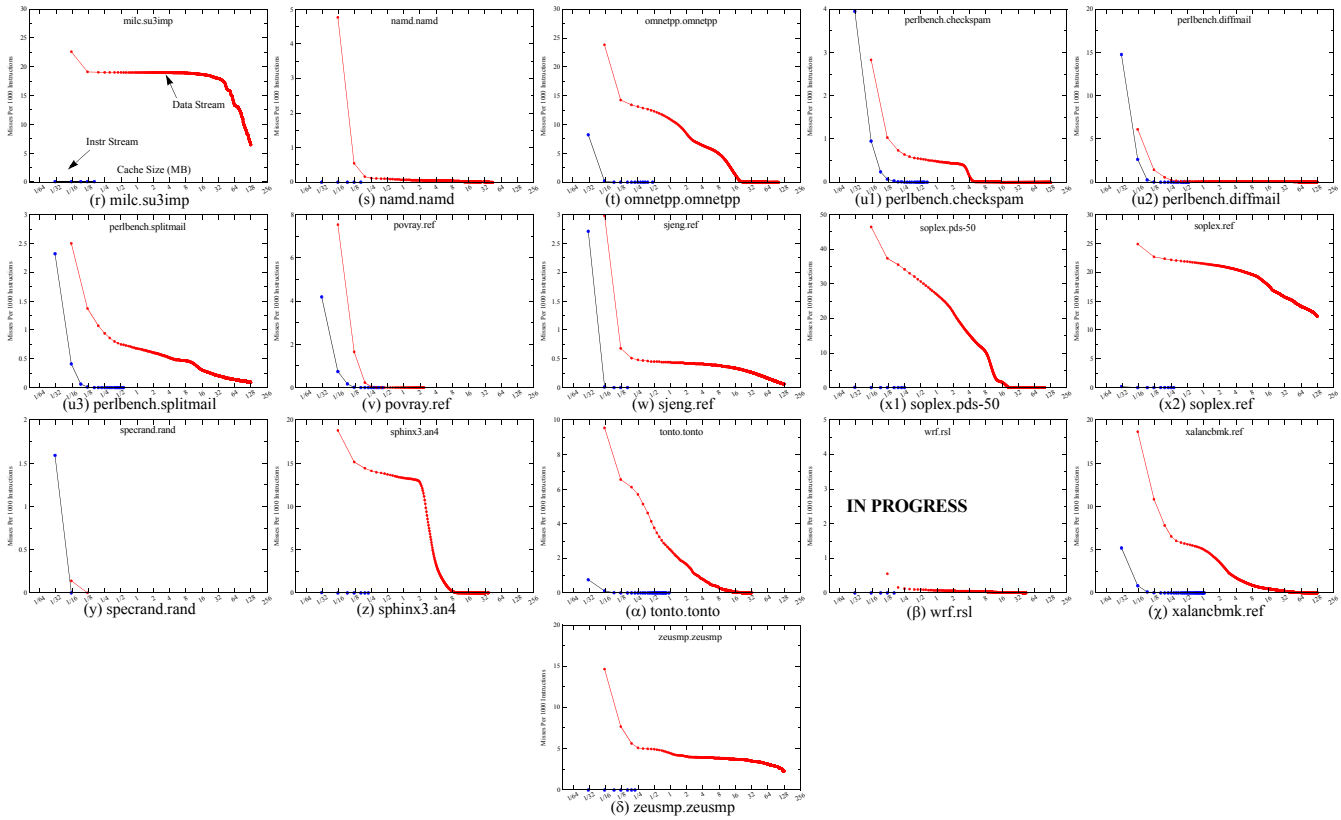


Figure 6: SPEC CPU2006 Cache Sensitivity: The figure shows the cache performance of each workload as a function of cache size. The y-axis presents workload MPKI and the x-axis presents the cache size in megabytes (MB). All workloads were run to completion using the reference input sets.

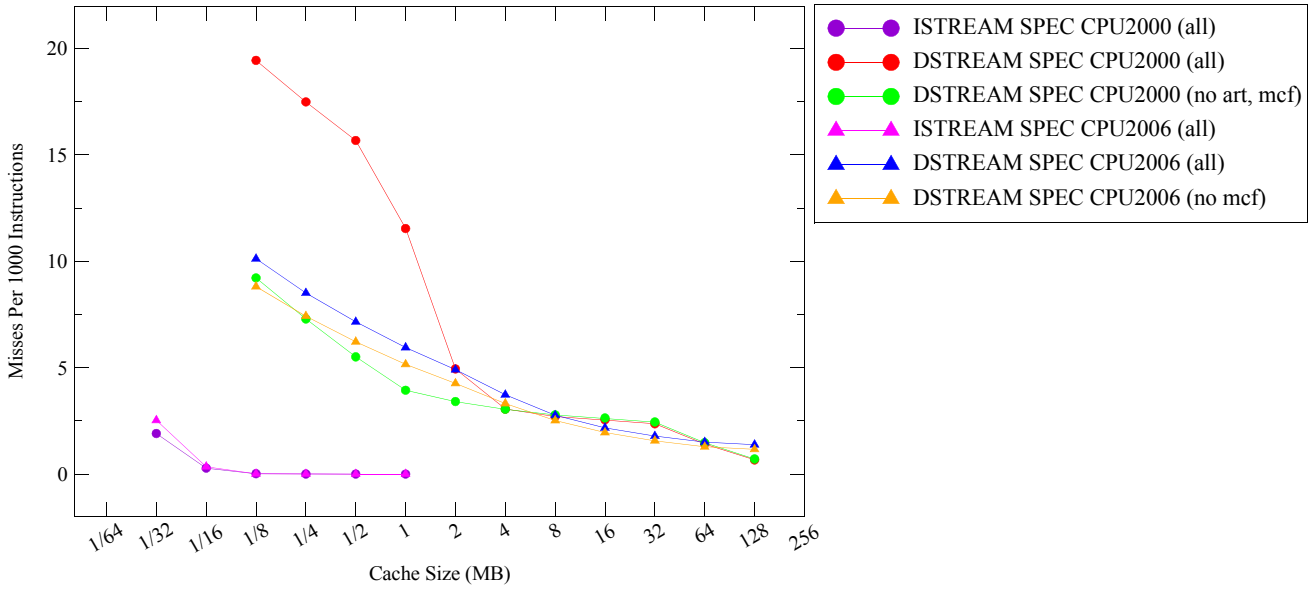


Figure 7: Average SPEC CPU2000 and SPEC CPU2006 Cache Sensitivity: The figure presents the average cache behavior of the SPEC CPU2000 and SPEC CPU2006 suites. Since SPEC CPU2000 is heavily biased by art and mcf, we also present the average behavior of the workloads excluding art and mcf. Similarly, for SPEC CPU2006 we present the average behavior excluding mcf.

100KB. However, for workloads such as *applu* (Figure 2a), *gap* (Figure 2l), and others where misses occur even in a 128MB cache, one can conclude that the total data memory footprint size of these workloads is greater than or equal to 128MB. From the figures, 20% of the 48 SPEC CPU2000 workloads have total memory footprint greater than 128MB while more than 75% of the 56 SPEC CPU2000 workloads have total memory footprint greater than 128MB.

The instruction cache sensitivity study reveals that an instruction cache that is 64KB suffices for most workloads in the two suites. However, there are some outlier workloads such as *crafty*, *gap*, *gcc*, *eon*, *perl* and *vortex* from SPEC CPU2000 and *gcc*, *gobmk*, *omnetpp*, *perl*, and *xalancbmk* from SPEC CPU2006 which require a 128KB instruction cache to perform as well as other workloads in the suite. These workloads have code footprints that are greater than 1MB. This is to be expected from workloads such as compilers (*gcc*), interpreters (*perl*), XML parsers (*xalancbmk*) and artificial intelligence programs (*gobmk* and *sjeng)* which spend their time over a wide range of functions.

The data cache sensitivity study reveals that most of the workloads exhibit cache friendly behavior—incremental increases in cache size yields incremental improvements in cache performance. Such behavior is observed when the cache miss rate shows a smooth exponentially decreasing behavior with increasing cache sizes (example Figure 2e). However, both suites also comprise of workloads where incrementally increasing the cache size provides no improvements in cache performance until a particular cache size is reached. At this cache size suddenly significant improvement in cache performance is observed. Such behavior is evident through sudden drops (cliff shaped figures) in the cache miss rate as a function of cache size. Some examples are *art* (Figure 2d) and *equake* (Figure 2h) from SPEC CPU2000 and *libquantum* (Figure 5p) and *sphinx* (Figure 6z) from SPEC CPU2006. This behavior implies that the respective workloads have a working set that is of the same cache size as where the significant drops in cache miss rate occur. For example, *libquantum* from the SPEC CPU2006 suite has the best cache performance with a 32MB cache. Since the total number of misses drops down to zero with a 32MB cache, this implies that the workload re-references a 32MB working set in a circular fashion. Some workloads like *gcc* in both SPEC CPU2000 and SPEC CPU2006 illustrate multiple drops in the cache miss rate function implying that they have multiple working set sizes.

Figure 7 presents the average cache behavior of both the instruction and data stream. Since the average behavior of SPEC CPU2000 is heavily biased by *art* and *mcf*, we also present the average of all workloads excluding *art* and *mcf*. Comparatively, for SPEC CPU2006 we also present the average of all workloads excluding *mcf. The* figure shows that both SPEC CPU2000 and SPEC CPU2006 have similar instruction cache requirements, with 64KB being optimal. On the other hand, the knee of the data stream cache miss rate function for SPEC CPU2000 occurs at a cache size of 1-2 MB, while that of SPEC CPU2006 occurs at a cache size of 16-32 MB—a factor of 16 higher. The larger cache requirements of these workloads continues to put pressure on improving the performance of the memory subsystem.

3.4.2.Sensitivity to Different Input Sets

For the SPEC CPU2006 workloads, the benchmarks with multiple input sets are: *astar* (2 input sets), *bzip2* (6 input sets), *gamess* (3 input sets), *gcc* (9 input sets), *gobmk* (5 input sets), *h264ref* (3 input sets), *hmmer* (2 input sets), *perlbench* (3 input sets), and *soplex* (2 input sets). We compare the different input sets using MPKI as the metric of comparison. Input sets that have the highest MPKI are considered more memory intensive than others.

For the *astar* benchmark, the *BigLakes2048* input is more data memory intensive of the two. For the *bzip2* benchmark, the different input sets have very similar instruction and data cache requirements with the *liberty* input set being more data memory intensive than the others. For the *gamess* benchmark, all input sets have very similar working sets and fit well into a 512KB cache (the *triazolium* input set is slightly more data memory intensive for a 256KB cache). For the *gcc* benchmark, the *166* and *g23* input sets are more data memory intensive than the others. For the *gobmk* and *h264ref* benchmarks, all input sets have very similar working set sizes. For the *hmmer* benchmark, the *retro* input set is more data memory intensive of the two. For the *perlbench* benchmark, the *splitmail* input set is more data memory intensive. Finally, for the *soplex* benchmark, the *pds-50* input set is more data memory intensive when the cache size is less than 4MB while the *ref* input set is more data memory intensive when the cache size is greater than 4MB.

3.4.3.Performance of a Three-Level Cache Hierarchy

Figure 8 presents the cache performance of the workloads for a three-level cache hierarchy: 32KB separate instruction and data caches, 256KB unified L2 cache and 2MB unified L3 cache. For each cache we present four metrics: *accesses per 1000 instructions*, *misses per 1000 instructions*, *miss rate*, and *evictions unused*.

The *evictions unused* metric is a measure of the total number of cache lines evicted that were never re-referenced after they were inserted into the cache. There are three main reasons for unused cache evictions. First, the referenced data has no temporal locality. This behavior is characteristic of references to streaming data. Second, the data has short temporal locality that is handled by smaller caches. In such scenarios, the filtering effect of the small caches causes the data to never be re-referenced in successive levels of the cache hierarchy. Finally, unused evictions also occur when the referenced data has a working set that is larger than the

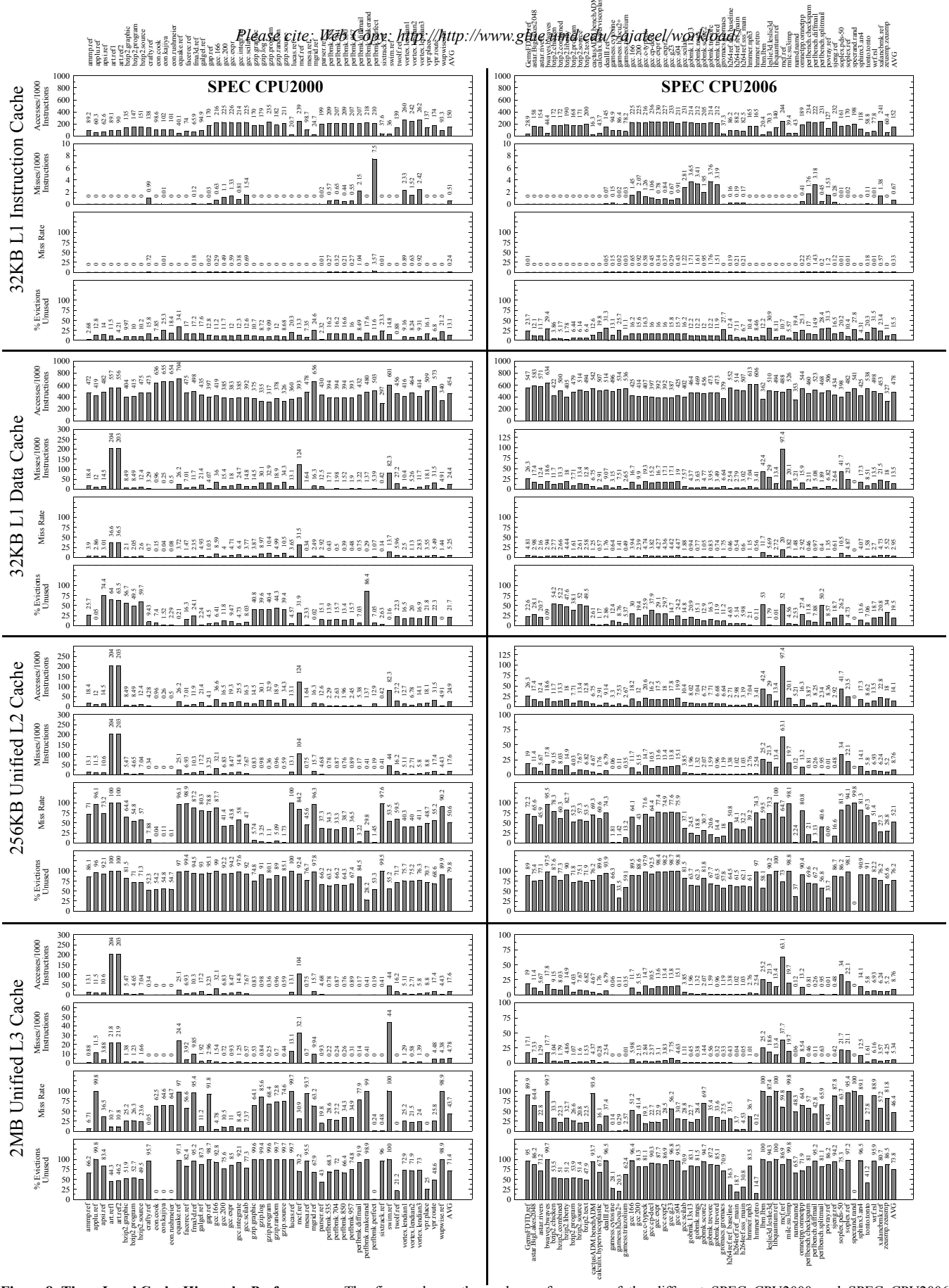


Figure 8: Three Level Cache Hierarchy Performance: The figure shows the cache performance of the different SPEC CPU2000 and SPEC CPU2006 workloads for a three level non-inclusive cache hierarchy

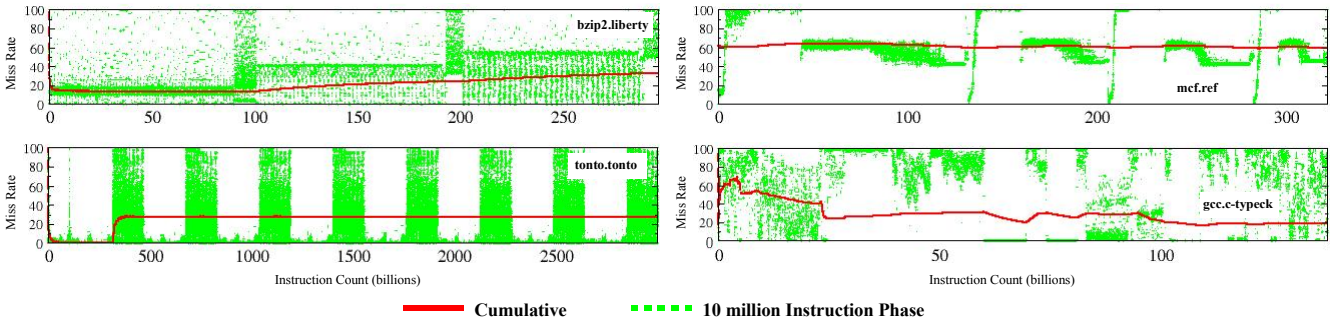


Figure 9: Full Run L3 Cache Behavior of Applications: The figure shows the full run L3 cache miss-rate behavior of workloads from SPEC CPU2006. The solid line presents cumulative behavior while the dots represent application behavior the on a 10 million instruction granularity.

available cache size. As a result, cache lines get evicted due to capacity misses before they ever get a chance to be reused.

The cache performance numbers presented in Figure 8 draws similar conclusions on the instruction and data cache performance as discussed in Section 3.4.1. The key observations in these figures is the *unused evictions* metric. The figures show that on average, roughly 15%, 20%, 80%, and 75% of the lines evicted from the IL1, DL1, UL2, and UL3 caches are never re-referenced after insertion into the cache. This presents a tremendous wastage of cache resources, especially in the large caches. Every line that is not re-referenced before eviction is wasted cache space that could potentially have been used by a more useful cache line. For applications with frequent cache misses, this suggests an opportunity to improve cache performance by effectively utilizing the cache resources.

3.5. Full Run Behavior of Applications

Figure 9 presents the time varying L3 cache performance of selected benchmarks from the SPEC CPU2006 suite. The x-axis presents the total number of instructions executed (in billions) and the y-axis presents the L3 cache miss rate. For each benchmark we present the cumulative L3 cache miss rate of the application (as represented by the solid line) and the instantaneous behavior of the application on a 10 million instruction interval (as represented by the dots).

The figure shows that applications have several different phases of execution. For example, *bzip2* has a step function with the L3 cache miss rate increasing over time; *mcf* has several phases of execution where none, some, or all cache references result in misses; *tonto* presents loop behavior with the cache performance significantly varying within each loop; finally *gcc* illustrates the L3 cache performance significantly varying during its full execution run.

The phase behavior of these applications show that it is extremely important that representative regions chosen for detailed simulation accurately reflect actual program behavior. For example, choosing a 100 million instruction interval starting at 525 billion instructions for *tonto* would be misleading for performance studies since the L3 cache performance in this phase inaccurately reflects the overall L3 cache performance of the application.

Choosing representative regions of long running programs for detailed execution becomes a challenging task as newer applications continue to emerge. Since IDS allows for quick full-run exploratory studies of applications, it serves as an excellent methodology to not only identify representative regions of a program but also to validate existing region selection algorithms [12].

4. Summary

There is a growing need for simulation methodologies to understand the memory system requirements of emerging workloads in a reasonable amount of time. This paper presents binary instrumentation-driven simulation (IDS) as an alternative to conventional execution-driven and trace-driven simulation methodologies. Since instrumentation driven simulation is *fast*, *robust*, and *simple*, users can characterize applications at MIPS speed. As a result, rather than studying only small regions of an application, users can now study applications run to completion.

We illustrate the benefits of IDS using Pin to characterize the SPEC CPU2000 and SPEC CPU2006 benchmark suites. In comparison to SPEC CPU2000, workloads in SPEC CPU2006 have an instruction count that is an order of magnitude larger than the SPEC CPU2000 workloads. Specifically, the average instruction count of SPEC CPU2000 was 131 billion while that of SPEC CPU2006 is 1.3 trillion. The large instruction counts of SPEC CPU2006 now present interesting challenges in choosing representative regions for detailed simulation. A full memory characterization study of both SPEC CPU2000 and SPEC CPU2006 reveal that SPEC CPU2006 workloads have larger memory working-set sizes with most memory-intensive workloads requiring more than 4MB of cache size. The larger cache requirements of these workloads continues to put pressure on improving the performance of the memory subsystem.

5. References

[1] SPEC CPU2006: http://www.spec.org/cpu2006/Docs/readme1st.html

[2] P. Bungale and C. K. Luk "PinOS".

[3] J. Edler and M. D. Hill. "Dinero IV Trace-Driven Uniprocessor Cache Simulator".

[4] J. L. Henning. "SPEC CPU2006 Benchmark Descriptions." In ACM SIGARCH newsletter, Computer Architecture News, Volume 34, No. 4, September 2006.

[5] A. Jaleel, R. S. Cohn, C. K. Luk, B. L. Jacob. "CMP$im: Using PIN to Characterize Memory Behavior of Emerging Workloads on CMPs", Technical Report - UMD-SCA-2006-01

[6] K. Lawton. Bochs. http://bochs.sourceforge.net.

[7] C. K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation." In Proceedings of Programming Language Design and Implementation (PLDI), Chicago, Illinois, 2005.

[8] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, and G. Hallberg. Simics: A full system simulation platform. IEEE Computer, 35(2):50-58, Feb. 2002.

[9] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. "Evaluation techniques in storage hierarchies." IBM Journal of Research and Development, 9, 1970

[10] S. Narayanasamy, C. Pereira, H. Patil, R. Cohn, and B. Calder. "Automatic logging of operating system effects to guide application-level architecture simulation." In SIGMETRICS, Saint Malo, France, 2006.

[11] H. Pan, K. Asanovic, R. Cohn, and C. K. Luk. "Controlling Program Execution through Binary Instrumentation." In Workshop on Binary Instrumentation and Applications (WBIA), St. Louis, MO, September 2005.

[12] E. Perelman, G. Hamerly, M. V. Biesbrouck, T. Sherwood, and B. Calder. "Using SimPoint for Accurate and Efficient Simulation". In SIGMETRICS, San Diego, CA, 2003.

[13] S. Sair and M. Charney. "Memory Behavior of the SPEC CPU2000 Benchmark Suite." IBM Thomas J. Watson Research Center Technical Report RC-21852, October 2000.

[14] Srivastava and A. Eustace. "ATOM: A System for Building Customized Program Analysis Tools", Programming Language Design and Implementation (PLDI), 1994, pp. 196-205.

[15] R. A. Uhlig. and T. N. Mudge. "Trace-driven Memory Simulation: A Survey", In ACM Computing Surveys, Vol. 29, 1997.